



Domain-Specific Languages and Haskell

Anthony M. Sloane

Programming Languages Research Group
Department of Computing, Macquarie University
Sydney, Australia

<http://www.comp.mq.edu.au/plrg/>

Outline

Implementing Domain-Specific Languages in Haskell featuring:

Basics: Values, functions and referential transparency

Higher-order functions

Abstract data types and pattern matching

Lazy evaluation and infinite data structures

Equational reasoning

For much more information on Haskell see <http://haskell.org>

Domain-Specific Languages (DSLs)

Domain-Specific Languages help you solve problems in a particular restricted problem domain.

Essentially DSLs are the programming language equivalent of using a small **specialised vocabulary** to talk about a particular area of endeavour. Aids communication with domain experts.

DSLs are contrasted with **General-purpose Programming Languages** (GPLs) like C++ and Java that serve every possible domain at the cost of complexity.

For much more, see “**When and How to Develop Domain-Specific Languages**” by Mernik, Heering and Sloane, ACM Computing Surveys, ACM Press, December 2005, pp 316-344.

Implementing a DSL

As an **Application Program Interface (API)** in a GPL. In some sense, the interface to any application library is a form of DSL.

By writing a **compiler** that translates the DSL programs into some other language.

By **embedding** the DSL inside a GPL. Similar to implementation as an API but if the appropriate kind of GPL is chosen then the expressive power can be increased.

We choose the last approach and use the functional programming language Haskell as the GPL.

Functional Programming Languages

Languages where computation is achieved by applying **functions** to values to get new values.

No variables: Once a value has been computed it cannot be changed.

Referential transparency: equals can be replaced with equals.

Equational reasoning: proving properties of programs.

For much more, see “**Conception, Evolution, and Application of Functional Programming Languages**” by Hudak, ACM Computing Surveys, ACM Press, September 1989, pp 359-411 and “**The Haskell School of Expression**” by Hudak, Cambridge University Press, 2000.

The Music domain

Suppose we want a [language for expressing computations on music](#).

We want to be able to:

- write expressions that stand for pieces of music, and

- write programs that compute with music or transform a piece of music into another, and

- prove properties of our programs.

Concepts in this domain include: time, note, pitch, rest, tempo, transposition, instrument, line, chord, duration, trill, and performance.

Basic Values and Operations

Basic values such as **integers** : ..., -2, -1, 0, 1, 2, ... and **characters**: 'a', 'b'.

Basic Values and Operations

Basic values such as **integers** : ..., -2, -1, 0, 1, 2, ... and **characters**: 'a', 'b'.

Basic operations provided by **pre-defined functions** such as (+), (*):

$$3 + 4 \Rightarrow 7$$

$$2 + 5 * 3 \Rightarrow 17$$

Basic Values and Operations

Basic values such as integers : ..., -2, -1, 0, 1, 2, ... and characters: 'a', 'b'.

Basic operations provided by pre-defined functions such as (+), (*):

$$3 + 4 \Rightarrow 7$$

$$2 + 5 * 3 \Rightarrow 17$$

User-defined values:

```
data PitchClass = Cf | C | Cs | Df | D | Ds | Ef | E | Es | Ff | F | Fs  
                | Gf | G | Gs | Af | A | As | Bf | B | Bs
```

Types

Haskell is a **strongly-typed** language, although type declarations can be omitted in many cases and the **types are inferred** by the compiler.

Basic types: ..., -2, -1, 0, 1, 2, ... :: Int
'a', 'b', 'c', ... :: Char

Types

Haskell is a **strongly-typed** language, although type declarations can be omitted in many cases and the **types are inferred** by the compiler.

Basic types: ..., -2, -1, 0, 1, 2, ... :: Int
'a', 'b', 'c', ... :: Char

User-defined types: Cf, C, ..., Bs :: PitchClass

Types

Haskell is a **strongly-typed** language, although type declarations can be omitted in many cases and the **types are inferred** by the compiler.

Basic types: ..., -2, -1, 0, 1, 2, ... :: Int
'a', 'b', 'c', ... :: Char

User-defined types: Cf, C, ..., Bs :: PitchClass

Type synonyms:

```
type Octave = Int
type Pitch = (PitchClass, Octave)
```

Parameterised Data Types

data Integral a => Ratio a = a :% a

Parameterised Data Types

data Integral a => Ratio a = a :% a

Example:

3 :% 4 :: Ratio Int

Parameterised Data Types

data Integral a => Ratio a = a :% a

Example:

3 :% 4 :: Ratio Int

Simple functions on Ratio can be defined by **pattern matching**:

numerator, denominator :: Ratio a -> a

numerator (x :% y) = x

denominator (x :% y) = y

Parameterised Data Types

data Integral a => Ratio a = a :% a

Example:

3 :% 4 :: Ratio Int

Simple functions on Ratio can be defined by **pattern matching**:

numerator, denominator :: Ratio a -> a

numerator (x :% y) = x

denominator (x :% y) = y

Example:

numerator (3 :% 4) => 3

denominator (9 :% 12) => 12

Reducing Ratios

Example:

$$9 : 12 \Rightarrow 3 : 4$$

Reducing Ratios

Example:

$$9 \% 12 \Rightarrow 3 \% 4$$

Code:

```
(%) :: Integral a => a -> a -> Ratio a  
x % y = reduce (x * signum y) (abs y)
```

Reducing Ratios

Example:

$$9 \% 12 \Rightarrow 3 \% 4$$

Code:

```
(%) :: Integral a => a -> a -> Ratio a  
x % y = reduce (x * signum y) (abs y)
```

```
reduce :: Integral a => a -> a -> Ratio a  
reduce x y | y == 0    = error "reduce: zero denominator"  
           | otherwise = (x `quot` d) :% (y `quot` d)  
                        where d = gcd x y
```

Structured Data Types

```
type Dur = Ratio Int
data Music = Note Pitch Dur
          | Rest Dur
          | Music :+: Music
          | Music :=: Music
          | Tempo Ratio Music
          | Trans Int Music
```

```
-- duration in whole notes
-- play a note
-- play nothing
-- play in sequence
-- play in parallel
-- scale tempo
-- transpose by an interval
```

Structured Data Types

```
type Dur = Ratio Int           -- duration in whole notes
data Music = Note Pitch Dur   -- play a note
              | Rest Dur       -- play nothing
              | Music :+: Music -- play in sequence
              | Music :=: Music -- play in parallel
              | Tempo Ratio Music -- scale tempo
              | Trans Int Music -- transpose by an interval
```

Examples:

```
(Note (A,4) (1%2)) :+: (Note (B,3) (3%4)) :: Music
```

```
wn = 1%1 :: Dur
```

```
hn = 1%2 :: Dur
```

```
Tempo (3%4) ((Note (C,4) wn) :+: (Rest hn)) :+:
              ((Note (D,4) hn) :=: (Note (E,4) hn)) :: Music
```

Duration of a Piece of Music

Define using [pattern matching](#) on the different music constructors:

```
dur :: Music -> Dur
dur (Note _ d)    = d
dur (Rest d)      = d
dur (m1 :+: m2)   = (dur m1) + (dur m2)
dur (m1 :=: m2)   = (dur m1) `max` (dur m2)
dur (Tempo a m)   = (dur m) / a
dur (Trans _ m)   = dur m
```

Examples:

```
hn + hn => wn
dur Tempo (3%4) ((Note (C,4) wn) :+: (Rest hn)) :+:
                ((Note (D,4) hn) :=: (Note (E,4) hn))
=> 8 % 3
```

Lists

A standard **recursive data type** used to represent many different kinds of data. Strings are lists of characters.

Pre-defined as:

```
data [a] = []           -- not legal Haskell syntax
          | a : [a]
```

Examples:

```
[]
1 : (2 : (3 : []))
[1, 2, 3]           -- syntactic sugar
'B' : 'o' : 'b' : [] => ['B','o','b'] => "Bob"
```

Mapping Lists

`map` is a standard example of a **higher-order function**: one that takes a function as an argument and/or returns a function as its result.

```
map :: (a -> b) -> [a] -> [b]
map _ []      = []
map f (x:xs) = (f x) : (map f xs)
```

Mapping Lists

`map` is a standard example of a **higher-order function**: one that takes a function as an argument and/or returns a function as its result.

```
map :: (a -> b) -> [a] -> [b]
map _ []      = []
map f (x:xs) = (f x) : (map f xs)
```

Examples:

```
map (+1) [3,5,7] => [4,6,8]
```

```
map chr [72,101,108,108,111] => "Hello"
```

Folding Lists

`foldr` is a more complex example of a higher-order function. In this case we use a function argument to combine the elements of a list.

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr f z [] = z`

`foldr f z (x:xs) = f x (foldr f z xs)`

Folding Lists

`foldr` is a more complex example of a higher-order function. In this case we use a function argument to combine the elements of a list.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Example:

```
foldr (+) 0 [1,2,3,4]
=> (+) 1 (foldr (+) [2,3,4]
...
=> (+) 1 ((+) 2 ((+) 3 ((+) 4 (foldr (+) 0 []))))
=> (+) 1 ((+) 2 ((+) 3 ((+) 4 0)))
=> 1 + 2 + 3 + 4 + 0
=> 10
```

Lines, Chords and Delays

Using foldr to compute with music.

```
chord :: [Music] -> Music  
chord = foldr (:=:) (Rest 0 % 1)
```

Lines, Chords and Delays

Using foldr to compute with music.

```
chord :: [Music] -> Music
chord = foldr (:=:) (Rest 0 % 1)
```

Example:

```
c o = Note (C,o); e o = Note (E,o); g o = Note (G,o)
cMaj = [ n 4 hn | n <- [c, e, g] ] -- list comprehension
cMajChd = chord cMaj
```

```
cMajChd =>
  Note (C,4) (1 % 2) :=: (Note (E,4) (1 % 2) :=:
    (Note (G,4) (1 % 2) :=: (Rest (0 % 1))))
```

More Building Blocks

General Haskell **library functions** are useful for building up functions to compute more complex forms of music.

```
repeat :: a -> [a]
```

```
repeat x = xs where xs = x:xs
```

```
-- an infinite list!
```

More Building Blocks

General Haskell [library functions](#) are useful for building up functions to compute more complex forms of music.

```
repeat :: a -> [a]
repeat x = xs where xs = x:xs           -- an infinite list!
```

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take (n-1) xs
```

More Building Blocks

General Haskell **library functions** are useful for building up functions to compute more complex forms of music.

```
repeat :: a -> [a]
repeat x = xs where xs = x:xs           -- an infinite list!
```

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take (n-1) xs
```

Example:

```
line = foldr (:+:) (Rest 0 % 1)
mkLn :: Int -> Pitch -> Dur -> Music
mkLn n p d = line (take n (repeat (Note p d)))
```


Equational Reasoning

The referential transparency of pure functional languages like Haskell means they have very precise **equational semantics**.

Thus we can **reason about programs** in a mathematical way.

This form of reasoning gives a much stronger **guarantee of correctness** than for traditional imperative languages where (partial) **testing** is about all you can do.

Concatenating lists conserves length

Want to show

$\text{length } (a++b)$

$= \text{length } a + \text{length } b$

for all lists a and b .

Concatenating lists conserves length

Want to show

$\text{length } (a++b)$

$= \text{length } a + \text{length } b$

for all lists a and b .

$\text{length} :: [a] \rightarrow \text{Int}$

$\text{length } [] = 0$

$\text{length } (x:xs) = 1 + \text{length } xs$

Concatenating lists conserves length

Want to show

$$\begin{aligned} \text{length } (a++b) \\ &= \text{length } a + \text{length } b \end{aligned}$$

for all lists a and b .

$$\begin{aligned} \text{length} &:: [a] \rightarrow \text{Int} \\ \text{length } [] &= 0 \\ \text{length } (x:xs) &= 1 + \text{length } xs \end{aligned}$$

$$\begin{aligned} (++) &:: [a] \rightarrow [a] \rightarrow [a] \\ [] \quad ++ \quad ys &= ys \\ (x:xs) \quad ++ \quad ys &= x : (xs \quad ++ \quad ys) \end{aligned}$$

Concatenating lists conserves length

Want to show

$$\begin{aligned} \text{length } (a++b) \\ &= \text{length } a + \text{length } b \end{aligned}$$

for all lists a and b.

$$\begin{aligned} \text{length} &:: [a] \rightarrow \text{Int} \\ \text{length } [] &= 0 \\ \text{length } (x:xs) &= 1 + \text{length } xs \end{aligned}$$

$$\begin{aligned} (++) &:: [a] \rightarrow [a] \rightarrow [a] \\ [] ++ ys &= ys \\ (x:xs) ++ ys &= x : (xs ++ ys) \end{aligned}$$

Base case:

$$\begin{aligned} \text{length } ([] ++ ys) \\ &= \text{length } ys \\ &= 0 + \text{length } ys \\ &= \text{length } [] + \text{length } ys \end{aligned}$$

Concatenating lists conserves length

Want to show

$$\begin{aligned} \text{length } (a++b) \\ &= \text{length } a + \text{length } b \end{aligned}$$

for all lists a and b.

$$\begin{aligned} \text{length} &:: [a] \rightarrow \text{Int} \\ \text{length } [] &= 0 \\ \text{length } (x:xs) &= 1 + \text{length } xs \end{aligned}$$

$$\begin{aligned} (++) &:: [a] \rightarrow [a] \rightarrow [a] \\ [] ++ ys &= ys \\ (x:xs) ++ ys &= x : (xs ++ ys) \end{aligned}$$

Base case:

$$\begin{aligned} \text{length } ([] ++ ys) \\ &= \text{length } ys \\ &= 0 + \text{length } ys \\ &= \text{length } [] + \text{length } ys \end{aligned}$$

Inductive case:

If

$$\begin{aligned} \text{length}(xs ++ ys) \\ &= \text{length}(xs) + \text{length}(ys) \end{aligned}$$

Then

$$\begin{aligned} \text{length}((x:xs) ++ ys) \\ &= \text{length } (x : (xs ++ ys)) \\ &= 1 + \text{length } (xs ++ ys) \\ &= 1 + \text{length}(xs) + \text{length}(ys) \\ &= \text{length}(x:xs) + \text{length}(ys) \end{aligned}$$

reverse doesn't change the length of a list

Want to show

$\text{length}(\text{reverse } l) = \text{length } l$
for all lists l .

reverse doesn't change the length of a list

Want to show

$\text{length} (\text{reverse } l) = \text{length } l$
for all lists l .

$\text{reverse} :: [a] \rightarrow [a]$

$\text{reverse } [] = []$

$\text{reverse } (x:xs) = \text{reverse } xs ++ [x]$

reverse doesn't change the length of a list

Want to show

$\text{length} (\text{reverse } l) = \text{length } l$
for all lists l .

Base case:

$\text{length} (\text{reverse } []) = \text{length } []$

$\text{reverse} :: [a] \rightarrow [a]$

$\text{reverse } [] = []$

$\text{reverse } (x:xs) = \text{reverse } xs ++ [x]$

reverse doesn't change the length of a list

Want to show

$\text{length} (\text{reverse } l) = \text{length } l$
for all lists l .

$\text{reverse} :: [a] \rightarrow [a]$

$\text{reverse } [] = []$

$\text{reverse } (x:xs) = \text{reverse } xs ++ [x]$

Base case:

$\text{length} (\text{reverse } []) = \text{length } []$

Inductive case:

If

$\text{length} (\text{reverse } xs) = \text{length } xs$

then

$\text{length} (\text{reverse } (x:xs))$

$= \text{length} (\text{reverse } xs ++ [x])$

$= \text{length} (\text{reverse } xs) + \text{length } [x]$

$= \text{length } xs + 1$

$= \text{length } (x:xs)$

A Final Example: Quicksort

The **expressiveness of Haskell** can often lead to very concise definitions of standard algorithms that mirror informal definitions. E.g., **Quicksort** sorts a list as follows:

- A sorted empty list is the empty list.
- A non-empty list is sorted by:
 - Picking an element of the list as a **pivot**.
 - Finding all of the elements smaller than the pivot and sorting those. Do the same for the elements bigger than the pivot.
 - Assemble the final sorted list by joining the sorted smaller elements to the pivot to the sorted bigger elements.

A Final Example: Quicksort

`qsort :: (Ord a) => [a] -> [a]`

A Final Example: Quicksort

`qsort :: (Ord a) => [a] -> [a]`

`qsort [] = []`

A Final Example: Quicksort

```
qsort :: (Ord a) => [a] -> [a]
```

```
qsort [] = []
```

```
qsort (pivot : rest) =
```

```
  qsort lower ++ [pivot] ++ qsort upper
```

A Final Example: Quicksort

`qsort :: (Ord a) => [a] -> [a]`

`qsort [] = []`

`qsort (pivot : rest) =`

`qsort lower ++ [pivot] ++ qsort upper`

`where lower = [x | x <- rest, x <= pivot]`

`upper = [x | x <- rest, x > pivot]`

Summary

Haskell's

simple **syntax**, and

convenient support for **abstract data types**, and

powerful ways to combine **higher-order functions**, and

equational semantics

makes it an excellent platform with which to experiment with DSLs in a verifiable way.